PR #26861 完整报告

sgl-project/sglang

[loader] Reduce transient allocations in NVFP4 MoE setup

合并时间：2026-06-04 12:13

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/26861

执行摘要

- 一句话：预分配 NVFP4 MoE 权重张量避免内存碎片
- 推荐动作：此 PR 值得精读，尤其是对内存在 GPU 上管理有优化兴趣的工程师。设计模式：避免临时分配列表再堆叠，而是预分配和重用缓冲区；条件跳过无关工作以减少内存峰值。

功能与动机

PR 提交信息指出：'Preallocate shuffled weight and scale tensors for TRTLLM FP4 MoE setup, and skip temporary blockscale swizzle placeholders when TRTLLM replaces them after weight loading. This avoids memory fragmentation and save a few GBs of HBM depending on the models.' 合并者 merrymercy 在提交消息中明确说明避免 GPU 内存碎片。

实现拆解

1. 预分配输出张量（utils.py）：将原先的 list + stack 模式改为先创建 torch.empty_like 的连续大张量，然后在循环中直接赋值子 slice，避免每个专家一次临时分配。
2. 引入可重用 scratch buffer：新增 _alloc_scale_buffers 辅助函数，一次性分配 scale 输出张量和 permuted 输入的暂存缓冲区（scratch）。循环中使用 torch.index_select 将 permuted 数据写入 scratch，然后调用 nvfp4_block_scale_interleave 直接输出到目标 slice，避免 .contiguous() 调用。
3. 条件跳过 blockscale swizzle（modelopt_quant.py）：在 ModelOptFP4Config.create_weights 方法中，当 self.enable_flashinfer_trtllm_moe 为 True 时，将 w13_blockscale_swizzled 和 w2_blockscale_swizzled 设为 None，而非直接创建 swizzle_blockscale 参数，因为 TRTLLM 会在 process_weights_after_loading 中替换该张量。
4. 格式化：第二个提交运行了 black 格式化。

关键文件：

- python/sglang/srt/layers/quantization/utils.py（模块 量化工具；类别 source；类型 core-logic；符号 _alloc_scale_buffers）：核心优化逻辑所在：重写 prepare_static_weights_for_trtllm_fp4_moe 函数，引入预分配和 scratch buffer，减少临时 GPU 分配。
- python/sglang/srt/layers/quantization/modelopt_quant.py（模块 量化配置；类别 source；类型 data-contract）：条件跳过 blockscale swizzle 分配，减少不必要的 GPU 内存占用

，配合 TRTLLM 的加载后替换。

关键符号：_alloc_scale_buffers, prepare_static_weights_for_trtllm_fp4_moe, ModelOptFP4Config.create_weights

关键源码片段

python/sglang/srt/layers/quantization/utils.py

核心优化逻辑所在：重写 prepare_static_weights_for_trtllm_fp4_moe 函数，引入预分配和 scratch buffer，减少临时 GPU 分配。

```
def _alloc_scale_buffers(scales):
    # 获取每个 expert 的 scale 输入形状和元素数
    per_expert_shape = scales[0].view(torch.uint8).shape
    per_expert_numel = scales[0].numel()
    # 预分配整个输出张量 (num_experts, per_expert_numel) 和一个可复用的 scratch 缓冲区
    output = scales.new_empty((num_experts, per_expert_numel), dtype=torch.uint8)
    scratch = torch.empty(per_expert_shape, dtype=torch.uint8, device=scales.device)
    return output, scratch

# 预分配 weight 和 scale 输出张量
# 原代码使用 list + torch.stack，每个 expert 都会产生一次临时分配
gemm1_weights_fp4_shuffled = torch.empty_like(gemm1_weights_fp4.view(torch.uint8))
gemm2_weights_fp4_shuffled = torch.empty_like(gemm2_weights_fp4.view(torch.uint8))
gemm1_scales_fp4_shuffled, g1s_scratch = _alloc_scale_buffers(gemm1_scales_linear_fp4)
gemm2_scales_fp4_shuffled, g2s_scratch = _alloc_scale_buffers(gemm2_scales_linear_fp4)

for i in range(num_experts):
    # ... 获取 permute_indices 和 permute_sf_indices 的代码保持不变 ...
    # 直接写入预分配张量的第 i 个 slice，避免 append + contiguous
    gemm1_weights_fp4_shuffled[i] = gemm1_weights_fp4[i].view(torch.uint8)[permute_indices.
    to(...)]

    # 使用 index_select 将 permuted 数据写入 scratch，然后 interleave 输出到目标 slice
    torch.index_select(
        gemm1_scales_linear_fp4[i].view(torch.uint8),
        0,
        permute_sf_indices.to(...),
        out=g1s_scratch,
    )
    gemm1_scales_fp4_shuffled[i] = nvfp4_block_scale_interleave(g1s_scratch)

    # 对 w2 同理
    gemm2_weights_fp4_shuffled[i] = gemm2_weights_fp4[i].view(torch.uint8)[permute_indices.
    to(...)]
    torch.index_select(...)
    gemm2_scales_fp4_shuffled[i] = nvfp4_block_scale_interleave(g2s_scratch)

del g1s_scratch, g2s_scratch
```

评论区精华

合并者 merrymercy 直接批准并触发了 rerun-test。AI 审核机器人给出了正面总结。无争议或未解决问题。

- 暂无高价值评论线程

风险与影响

- 风险：主要风险在于逻辑等价性：原先 list+stack 后 contiguous()，新方式直接赋值 slice 可能改变内存连续性假设，但预分配的大张量本身是连续的，slice 也是连续部分，且 TRTLLM 内核期望连续的 uint8 张量，风险较低。当 enable_flashinfer_trtllm_moe 为 False 时，行为保持不变；只有 True 时才跳过 swizzle，不会影响非 TRTLLM 路径。潜在性能风险：如果 scratch buffer 在不同 CUDA 流上导致同步问题，但所有操作都在同一流上且循环串行，应该没问题。缺少直接单元测试覆盖，但已有集成测试覆盖 FP4 模型加载。
- 影响：影响范围限于使用 TRTLLM FP4 MoE 量化的大模型（如 DeepSeek V3）。减少内存碎片可能使更大规模的模型部署或更大的 batch 成为可能。无用户 API 变化，属于后台优化，对开发者和运维透明。
- 风险标记：缺少单元测试覆盖，TRTLLM 特定优化

关联脉络

- 暂无明显关联 PR